1 Overview

Today we’re starting a completely new topic, which is the external memory model, also known as the disk access model. One of the earliest works on this was by Aggrawal and Vitter [1].

We’ve talked about streaming, which is where we cannot store all of the data. Here we imagine instead that our data cannot fit inside RAM, but we can store it in some other disk space. We assume that disk is infinite, but memory is finite.

In the RAM, we assume that memory of size $M$ is divided into pages of size $B$, i.e. there are $M/B$ pages. The unit here is arbitrary, each item takes up constant size. Similarly, there are infinite pages of size $B$ on disk. When we do memory I/O, we imagine that the cost is free, but when we do disk I/O, we imagine that we can fetch a page, which has a unit cost.

This makes practical sense since memory is orders of magnitude faster than hard drive space, and is often far smaller. Hard drives are usually much faster when reading sequentially, which is why we use the idea of pages.

In terms of exact numbers, hard drives support somewhere around 140MB/sec of sequential reads/writes, but only 120I/O’s/sec when doing reads/writes on random 4KB chunks. This is because the disk must physically spin to the read/write location. RAM supports somewhere around 5-10GB/sec of reads/writes, with much less penalty for random access. SSDs/Flash is somewhere in the middle, but with the issue that each index can only be written to a limited number of times.

In the future, we’ll work with the idea of algorithms that work even without knowing the exact values of $M$ and $B$, which are known as cache-oblivious algorithms.

2 Algorithms

2.1 Reading $N$ Items (Streaming)

It takes exactly $\lceil N/B \rceil = O(N/B + 1)$ cost to read $N$ items. Same for any streaming algorithm, since we ignore in-memory operations.

In general, we assume that we can choose the layout of the items on disk, in whatever manner is most convenient.
2.2 Matrix Multiplication

Given two $N \times N$ matrices $X$ and $Y$, we want their product. We divide $X$ and $Y$ into blocks of size $\sqrt{M}/2$ and $\sqrt{M}/2$, so that each block fits in memory. We then store each block contiguously in row/column order form in disk. If the input is not already in this form, we can rearrange it into this form, which has some complexity depending on the original input format, but again, we choose the original layout, so we can ignore this.

Examine each such block in the output matrix $X \cdot Y$. The resulting block is the sum of products of several pairs of blocks, going across matrix $X$ and down matrix $Y$. Then there are $(\frac{N}{\sqrt{M}})^2$ output blocks that need to be produced. Each output block requires multiplying $O(\frac{N}{\sqrt{M}})$ pairs of blocks, where each chunk can be read with $O(M/B)$ I/O's. The total number of I/O's required is $O(\frac{N^3}{\sqrt{M}} \cdot M/B) = O(\frac{N^3}{B \sqrt{M}})$. This can be improved with better matrix multiplication.

2.3 Linked Lists

Dynamic vs. Static Data Structures. A data structure problem is one where you can do updates and queries to some data. A static data structure is one where the input data stays unchanged over time, i.e. no updates.

Linked lists are dynamic and must support insert($x$, $p$), where $p$ is a pointer to an entry you want to insert after. There is similarly remove($p$), which requires removing the item pointed to by $p$. There is also traverse($p$, $k$), where the $k$ elements after $p$ must be touched, i.e. print out their values.

Clearly the standard linked list solves these three operations optimally. The updates are $O(1)$, and traverse takes $O(k)$ time.

One idea is to block up nodes into chunks of size $B$. If this were static, this would satisfy traverse easily since this is essentially an array. However, how do we update these over insertions and deletions? While updating/deleting, we can maintain the invariant that all groups must have size at least $B/2$, except possibly the very last group.

When deleting, we can usually just delete the element from the one group. If this pushes its size less than $B/2$, we can merge with an adjacent group. If this pushes the adjacent group’s size above $B$, so we can then split evenly into size between $B/2$ and $3B/4$ elements. This will take two disk operations. All of the rearrangement logic here is free because it can be done in memory.

When inserting, we insert into the block. Again, if this pushes the group’s size above $B$, we can split evenly into two. This again takes two disk operations.

Traverse works as expected, going through blocks until $k$ elements are seen. This requires reading at most $2k/B$ items, since each cost corresponds to reading at least $B/2$ items.

2.4 Predecessor

Here we have $N$ static items, which are ordered and comparable. Each item is a (key,value) pair, where the keys are being compared. Then we need to support query($k$), where the predecessor item must be returned, i.e. the item with greatest key less than key $k$. For example, keys could be IP
addresses, and queries in a router table would return where to forward the IP address.

We can also have the same problem, except dynamic, i.e. there are inserts and deletes.

Static predecessor can be solved by storing a sorted array in memory, i.e. in $N$ space, with $O(\log N)$ queries using binary search. On an actual computer, which follows the word RAM model, we can perform operations on actual bits, with word size $W$, usually $W = \Theta(\log N)$. Predecessor has been perfectly well understood. We can get $O(N)$ memory with $O(\log \log N)$ queries, with Van Emde Boas trees [4]. With $O(N^{O(1)})$ memory, say $O(N^{1.01})$, we can have $O(\log \log N/ \log \log \log N)$ query time. This was shown by Beame and Fich [3]. These bounds have been proven optimal by Patrascu and Thorup [7].

### 2.5 (a,b)-tree / B-tree

In an external memory model, we can solve this problem with $(a,b)$-trees, which are used in databases and were invented by Bayer and McCreight [2]. This is kind of like a binary search tree, except each internal node has within $[a,b]$ children (except possibly the root, which has either no children or $\geq 2$). Each node, regardless of whether it is a leaf or an internal node, stores an array of $B$ values. Items are stored in leaves. Guides are stored in internal nodes. Each guide at index $i$ in an internal node corresponds to a dividing value between $i$ and $i + 1$, where all values in the subtree at $i$ must have value less than the guide, and the subtree at $i + 1$ must have value $\geq$ than the guide.

Every time an insert happens, traverse the tree downwards, and place the new value into the leaf where it should go. If the leaf overflows, split the leaf into two leaves, and copy one of its keys to the parent. If that parent overflows, split that, etc., up until the root.

On removal, when a leaf is removed, the node may underflow (go under $a$). If so, merge it a neighboring (sibling) leaf, which removes one guide from its parent. This may cause another underflow, another merge, etc., up until the root. Then we have a guarantee of at least $a$ children, so the height is limited by $O(\log_a(n/a))$, which is usually $O(\log n)$ since $a$ is chosen to be a constant. There is an additional computation time of $\log b$ per operation, since we need to binary search the arrays, but $b$ is usually chosen proportional to $a$.

A B-tree is just a $(B/2, B)$-tree, so that we can fit each node in one block. Thus each operation takes at most one I/O per height, which is $O(\log_B N)$.

### 2.6 (2, 4)-tree with Buffers / Buffered Repository Tree (BRT)

**Tradeoffs.** Compare the B-tree to a logging model, where we just log all operations and scan through them. This model has query cost $O(N/B + 1)$. An insertion, which just appends one log entry, takes at most 1 disk update. Amortizing this across all inserts, however, only every $B$ inserts needs to actually update the disk if we cache in memory. Thus, updates take amortized $1/B$ cost.

Can we get a better insertion time while still keeping the better query time? We can, by using a $(2, 4)$-tree with buffers, shown by Buchsbaum et al. [6]. For each node, keep a buffer. When inserting, simply append to the root’s buffer. When the buffer becomes full, then flush it to its children. When the buffer at the leaf is full, then split the leaf, and apply the normal promotion operation.
Each query has the same cost, since we are just reading the buffer along with its node. This takes \( O(1) \) I/Os per height.

It turns out that the amortized complexity of insertion/deletion is \( O(\log N/B) \). We can imagine allocating each insert \( \theta(\log N/B) \) dollars, where 1 dollar can pay for 1 disk I/O. Since items are only flushed downwards, we can charge each item \( 1/B \) dollars. The number of times an item can be flushed is limited by the height, \( O(\log N) \), so we have just enough money.

The given scheme has a poor worst-case complexity, since it could potentially recursively flush to both sides. Instead, when flushing, we can choose to only flush to the child that would receive the most items, which flushes at least \( B/4 \) items. This maintains the amortized complexity above, but decreases the worst case to \( O(\log N) \).

An improvement by Brodal and Fagerberg [5] is to use a \((B^\epsilon, 2B^\epsilon)\) tree with buffers, called a buffered B-tree. The resulting height is \( \log_{B^\epsilon} N = \frac{1}{\epsilon} \log_B N \), and we have amortized insertion time \( \frac{1}{\epsilon B^\epsilon} \log_B N \), which is better than the standard \( B - \) tree.

### 2.7 Sorting

\( O(n \log n) \) is optimal for sorting in an comparison model. This is, however, not optimal in the word-RAM model, where sorting can be done in \( O(n \log \log n) \) time with Han’s sorting algorithm, and \( O(n \sqrt{\log \log n}) \) with randomization.

In the external memory model, given \( N \) elements, divide them into \( M/B \) groups, so that we can store one page from each group in memory. Merge sort will then consume \( M/B \cdot B = M \) memory at any time. Then we have the recursion \( T(N) = 1 \) for \( N \leq B \), and otherwise \( T(N) = O(N/B) + M/B \cdot T(N/(M/B)) \). Solving with the Master theorem, we have \( O(\frac{N}{B} \log_{\frac{M}{B}} N) \) as the final complexity.

This can be shown to be optimal in the comparison model. First assume that in the input, each block of \( B \) elements is sorted, since we might as well iterate through them and sort them in memory. Then the number of valid permutations is \( N!/(B!)^{N/B} \), which has a log of \( N \log N - \frac{N}{B} \log B = N \log \frac{N}{B} \). Given that we can only store \( M \) elements in memory, upon reading \( B \) elements, we can only learn how their interleave within the \( M \) elements. As a result, the amount of information we can gain is \( \log((\frac{M+B}{B})) = B \log \frac{M}{B} \). Then, dividing the information needed by that gained per I/O, we have a lower bound of \( \Omega(\frac{N}{B} \log_{\frac{M}{B}} N) \), as shown earlier.

We will learn next week about cache-oblivious versions of these algorithms and data structures, which will have the same performance for B-trees, linked lists, and sorting.

### References


